Atty. Dkt. No.: 037145/901 (NC 43838)

U.S. PATENT APPLICATION

for

SYSTEM AND METHOD FOR DYNAMICALLY ADDING FEATURES TO SOFTWARE APPLICATIONS

Inventors: Matti Pärnänen
Jari Laaksonen
Sami Rosendahl

As Assignors to:
Nokia Corporation
P.O. Box 226
FIN-00045 NOKIA GROUP
Finland

SYSTEM AND METHOD FOR DYNAMICALLY ADDING FEATURES TO SOFTWARE APPLICATIONS

FIELD OF THE INVENTION

[0001] The present invention relates to adding software application features to a device.

BACKGROUND OF THE INVENTION

[0002] Terminal software products are available that enable terminal manufacturers to create application-driven phones. The Series 60 Platform available from Nokia Corporation of Finland is an example of a terminal software product that can be licensed by terminal manufacturers.

[0003] A manufacturer that desires to make a phone (or other similar device) using a terminal software product includes various associated applications with the terminal software product. The manufacturer can also include user interface elements that have been customized by the manufacturer, add new applications or remove old ones. Further, a user purchasing such a phone can install third party software on the phone.

[0004] Typically, an application provides the functionality for a feature that is visible to an end user in a pre-defined shell application. The application menu and other user interface elements, like soft key buttons or selection lists, offer sets of choices or data available per feature. Nevertheless, manufacturers, end users, and third party developers cannot easily extend terminal software product applications with their own features in such way that extensions look integrated for

a seamless end user experience. Feature specific menu items (or other user interface elements) may not look and behave as if they were part of the terminal software product applications. Conventionally, to add a new feature into a phone means adding a new application, which generally leads to many applications doing the same kinds of things which are not necessarily consistent with each other.

[0005] Prior attempts at solving these problems have failed. For example, prior attempts have included implementing an application programming interface (API) for each feature. In such a system, an API is specified for each feature. Client applications needing the features are given a dependency to the API corresponding to the feature. Feature-specific UI elements are added to the client. Additionally, feature-specific UI elements can be added. Another attempt at solving the problems above includes adding feature-specific UI elements beforehand and using a run-time variation of the elements. Such a system contains all the features beforehand and features are activated/deactivated depending on the product configuration.

[0006] Nevertheless, these techniques typically cause static dependencies in software architecture, resulting in integration problems with phone development programs.

[0007] In any software development (including programming for terminal software products such as those described above), developers create a software architecture, map requirements to subsystems and components, and define functional and non-functional software requirements for the created architecture. Key components of these architectures are application programming interfaces (APIs), which represent interfaces via which another component, referred to as a caller, utilizes or calls another component, referred as callee.

[0008] To add a functionality into existing software, a new component is integrated into an existing application via the API the callee offers for others to use. In general, the API consists of set of declarations and definitions, e.g. constant declarations, type definition, class definitions, function (procedure) definition. Each function definition has a unique signature which consists of the return type, function name, and list of function arguments. Each argument defines a piece of data that is passed into a function or program. The data type that the argument can hold can be simple, like string, byte, integer, real, double, or structural reference to structure or class definition.

[0009] Standard, multi-purpose APIs may be needed in software development which are kept unchanged and generic enough over software evolution. Into such APIs, developers want to implement generic, multipurpose functions which take simple data types, such as Int genericDoSomething (const char[ ] objectInformation), for example. However, it can be difficult to implement generic functions which can accept many kind of parameters. The function can be separated into pieces to have separate variants. For example, the function can be defined as follows:

1) Int genericDoSomething (const char[ ] objectName);

2) Int genericDoSomething (const char[ ] objectId);

2B) Int genericDoSomethingByName (const char[ ] objectName);

2C) Int genericDoSomethingById (const char[ ]

objectId);

3) Int genericDoSomething (const char[ ] objectName, const char[] objectUrl);

4) Int genericDoSomething (const char[ ] objectName, const char[] objectID, char * objectUrl);

5) Int genericDoSomething (const char[ ] objectName, char[ ] outObjectInfo);

[0010] In all these versions, the callee assumes the meaning or semantics of each argument position. However, the callee must trust the caller and assume that the first input parameter is correct. The callee has no good way verify this. Alternatives 1) and 2) (above) are not possible because the signature of the function is not unique and the compiler cannot resolve the function call. Functions 2B) and 2C) are needed to resolve the function call.

[0011] When a callee passes an input parameter on to other functions, it delegates this trust onwards. After many layers of function calls, semantics may be lost and there is a risk that the argument is mis-understood.

[0012] If new arguments are added to a function, the signature of the function must be altered every time. Careless change might cause binary or source compatibility breaks. For example, in cases 3) and 4), additional objectID and objectUrl arguments are added. In case 5), if the callee returns data in output argument outObjectInfo, the caller faces similar problems with the input parameters. If the input argument is saved persistently into a file, the meaning of the argument

is lost. If the input argument is passed across process boundaries, the meaning of the argument is lost.

[0013] As mentioned above, it is possible to use specific function variants instead of a generic form. Semantic information can be added to an argument, such as "Int genericDoSomething (int objectInfoMeaning, const char[ ] objectInformation)." This applies to each additional argument passed in the same call (e.g., "Int genericDoSomething (int objectInfoMeaning1, const char[ ] objectInformation1, int objectInfoMeaning2, const char[ ] objectInformation2);"). The structures could be simplified (e.g., "Int genericDoSomething(struct ObjectInfo1 info, struct ObjectInfo2 info)"). Here, the structure "struct ObjectInfo1" holds the information as member data. In such a structure, the compiler checks semantics of the arguments.

[0014] Alternatively, classes can be used (e.g., "Int genericDoSomething(ObjectInfo1 info, ObjectInfo2 info);") where Object1 and Object2 are C++ classes which hold semantic information automatically. As such, the compiler checks semantics of the arguments. However, new arguments need to be added in a similar manner. Also, a genericDoSomething method can be added as a member function to the Object1 class to get object1.genericDoSomething(Object2 &info).

[0015] Another known technique (e.g. in Microsoft OLE2) is the usage of variant data type which can hold several data representations (see next section). But it still lacks the semantic information.

[0016] To pass parameters from a consumer to one or many matched providers through the same consumer API, a generic parameter with semantic and data information can be created. Each

consumer can, for example, then check the semantic meaning and data type, and make necessary conversion if needed. Parameters should be generic because it is difficult to know what type of data is passed through the API.

[0017] Thus, there is a need for providing a mechanism for adding features to an existing group of applications in a controlled and dynamic way. Even further, there is a need to add features into existing applications to extend the functionality of terminal software product applications. Yet further, there is a need for generic function arguments.

SUMMARY OF THE INVENTION

[0018] The present invention is directed to a method, device, system, and a computer program product where features are added dynamically to a software application. These features are added using a framework for a general unchangeable application programming interface (API) that adds any feature to any application.

[0019] Briefly, one exemplary embodiment relates to a method for adding computer software features dynamically to a software application by establishing a framework for a general, unchangeable application programming interface (API) that adds any feature to any application. The method includes providing a consumer application interest resource for a consumer application specifying desired user interface elements, storing the desired user interface elements corresponding to the consumer application interest resource in a file, communicating the desired user interface elements to an application interworking framework, and adding the desired user interface elements to a user interface associated with the software application.

[0020] Another exemplary embodiment relates to a device that adds features dynamically to a software application such that any feature provided by a software program can be added to a software platform program for the device. The device includes a new consumer application that publishes a feature interest indicating what features the new consumer application desires to have. The new consumer application provides user interface resources needed with interest placeholders for needed user interface features based on the feature interest. The device further includes multiple provider applications that have features and user interface resources available for the feature, and an application interworking framework that provides an interface for the new consumer application and the multiple provider applications such that the feature interest is matched with one of the features available from the multiple provider applications. The user interface elements corresponding to the matched feature are added from one of the multiple provider applications to the new consumer application.

[0021] Yet another exemplary embodiment relates to a system for adding features dynamically to a software application. The system includes a consumer application that publishes a feature interest and identifies user interface resources needed based on the feature interest, a provider application that publishes a provider capability and identifies user interface resources available for a feature, and an application interworking framework that provides an interface for the consumer application and the provider application such that the feature interest is matched with the provider capability and the user interface elements are added from the provider application to the consumer application.

[0022] Even another exemplary embodiment relates to a computer program product that has computer code to provide a

consumer application interest resource for a consumer application specifying desired user interface elements, store the desired user interface elements corresponding to the consumer application interest resource in a file, communicate the desired user interface elements to an application interworking framework, and add the desired user interface elements to a user interface.

[0023] Other principle features and advantages of the invention will become apparent to those skilled in the art upon review of the following drawings, the detailed description, and the appended claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] Exemplary embodiments will hereafter be described with reference to the accompanying drawings.

[0025] FIG. 1 is a diagrammatic representation of a software architecture to dynamically add features to software applications in accordance with an exemplary embodiment.

[0026] FIG. 2 is a diagrammatic representation of example user interfaces in accordance with an exemplary embodiment.

[0027] FIG. 3 is a flow diagram depicting operations in a process of dynamically adding features to software applications in accordance with an exemplary embodiment.

[0028] FIG. 4 is a diagrammatic representation of a generic parameters implementation in accordance with an exemplary embodiment.

[0029] FIG. 5 is a block diagram of a device according to an exemplary embodiment.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

[0030] FIG. 1 illustrates a software architecture to dynamically add features to software applications. A consumer application 12 indicates features available from feature providers 14 that the consumer application 12 has interest in by publishing a consumer interest 16. The consumer application 12 specifies the user interface (UI) element templates in a consumer UI resource file 18 and the feature providers 14 add the desired UI elements to the UI at run time.

[0031] In the situation where there is a feature provider 14 capable of fulfilling the published interest of the consumer application 12, the feature provider 14 adds UI elements to the UI at run time. Further, the feature provider 14 handles related business logic.

[0032] The consumer interest 16 and a provider capability 20 published by the feature provider 14 are expressed in the same, specified format. In an exemplary embodiment, this format includes multi-purpose Internet mail extensions (MIME) or Internet data, commands, and a provider interface. The consumer interest 16 and the provider interest 20 can include wild-cards patterns, which allows for more flexible resolving for the feature providers 14.

[0033] The data agreement between the consumer application 12 and the feature provider 14 to permit the communication of parameters from consumer to provider and vice versa. The data agreement can be static or it can be negotiated at run time. One consumer application 12 can use many providers and one feature provider 14 can fulfill many consumers' interests. The feature provider 14 can also be a consumer for other providers. The consumer application 12 and

the feature provider 14 are as independent as possible from each other. The feature provider 14 can be loaded only from read-only memory (ROM) or also from user data area (installable providers).

[0034] By way of example, the cases shown in Table 1 below are examples of the use of the software architecture described with reference to FIG. 1. Each example has an application, a consumer, an interest, and a provider.

Application	Consumer	Interest	Provider
Edit image	Image Viewer, Media Gallery	"Edit command" + MIME type (image/jpeg)	Image Fun
"Select application or function to be launched"	Softkey selector, Voice commands, Pinboard, Active Desk	"Select command" + MIME type (application object)	Application / Application manager
"Offer command options for found items in a text"	Browser, Messaging applications, Calendar, Notebook, ...	"Edit/View command" + Item(phone number, email address, URL, ...)	Browser, Phone, Phonebook, SMS Editor, MMS Editor, EMail Editor...
Fetch new image, audio, file	MMS Editor, Media Gallery	"New command" + MIME type (image/*, sound, video,file)	Camera, CamCorderMedia Gallery

Table 1

[0035] Thus, as shown in Table 1, the application "Edit image" has a consumer application of "Image Viewer" and "Media Gallery." Further, the corresponding consumer interest is "Edit command" and MIME type and the corresponding provider is "Image Fun." FIG 2 illustrates user interfaces in the case of an edit image application. In user interface 34, a new image fun feature 36 is added to an existing application menu in user interface 32. The addition of the new feature 36 is done in a seamless manner. As such, it seems to the user that the new feature is just another feature of Image viewer application.

[0036] Referring again to FIG. 1, the consumer application 12 owns a normal UI element in the consumer UI resource file 18. The consumer UI resource file 18 can include a consumer interest therein. Moreover, separate resource files are used to create collections of common interests or common interest "libraries." Wanted interests (e.g., menu commands, settings pages) can be inserted into wanted UI elements in the consumer UI resource file 18.

[0037] An Application Interworking Framework (AIWFW) 22 assures that the feature provider 14 can add provider UI elements 24 only in places where UI designers define UI elements. The AIWFW 22 establishes a connection between the consumer application 12 and the feature provider 14. The AIWFW 22 makes this connection by matching published consumer interests 16 with published provider interests 20. These connections can be pure dynamic link library (DLL) function calls from the consumer application 12 to a dynamically-loaded provider DLL, or they can have a inter-process communication (IPC) connection from consumer process to provider process. In an exemplary embodiment,

dynamically loaded DLLs are used to avoid too many context switches between processes.

[0038] The AIWFW 22 implements two APIs: one consumer API 26 for consumers to use providers and set of provider APIs 28 to implement providers. Feature providers 14 can implement only APIs that match their UI capability, e.g. base interface, menu service interface, settings interface, etc. The set of interfaces can be extended upon demand.

[0039] In operation, an interest is provided to the AIWFW 22. Based on data agreement, the consumer application 12 gives optional input parameters to the feature provider 14 and expects output parameters back, if feasible. The default formats of input and output parameters are sufficient in most cases.

[0040] Each feature provider 14 implements supported interfaces of the provider APIs 28. The AIWFW 22 offers default implementation for each interface. A first task for the feature provider 14 is to communicate to the consumer application 12 needed feature's UI elements from a provider resource file (PUI) 30, like menu items. A second task is to handle commands related to the added elements which appear in an integrated manner in the consumer resource file 18 along with native consumer items. A third task is to fulfill a data agreement, such as a default agreement.

[0041] The feature provider 14 uses any existing operating system APIs, user interface (UI) utilities etc. to help in implementation. The AIWFW 22 helps to implement feature providers easily in order to wrap-up existing, tested, software in the form of a feature provider. From

a security point, "Load-only-from-ROM" can be supported to restrict key providers to be non-patchable from user data area.

[0042] FIG. 3 illustrates exemplary operations performed in the dynamic addition of features to software applications process. Additional, fewer, or different operations may be performed, depending on the embodiment. In an operation 40, an interest resource is added for a consumer. The interest resource describes what features the consumer application desires.

[0043] In an operation 42, the user interface (UI) elements of the interest are provided into a resource file. Menu commands and settings pages are examples of the UI elements designated by the contents of the resource file. In an operation 44, a consumer application programming interface (API) provides the consumer interest to the AIWFW. The AIWFW is the interface between the consumer and the feature provider. In an operation 46, the feature provider adds the needed interest UI elements from the AIWFW to the consumer applications UI at run time.

[0044] The exemplary embodiments described with reference to FIGs. 1-3 extend applications in an integrated manner after the phone has been launched to market. The UI consistency of native platform applications can be maintained while specifying where additions in UI elements can take place. The whole platform becomes more interesting and customizable.

[0045] Accordingly, the platform configuration can be supported while creating different sales packages based on feature providers. At the same time, native applications can accept providers implemented afterwards by operators and 3rd developers.

[0046] From a software architecture point of view, many advantages can be seen. For example, maintenance and extension of features is easier. One feature provider can be customized and it impacts all the interested consumers in the similar manner. Thus, development costs are reduced. Moreover, architecturally static dependencies can be replaced with dynamic ones. This enables software configurations easier, e.g. implementing common core image and variant image concepts. Further, the exemplary embodiments allow easier device integration for phone development due to the incremental model for integration based on features. This should save integration burden and costs. Even further, new features can be added to the consumer application without changing (editing the code of) the consumer application. This reduces required testing effort considerably, The AIFWW implements the Open/Closed principle of Object-Oriented software design which has many good properties.

[0047] Feature variation is easier with the exemplary embodiments. The manufacturer can select wanted providers into ROM and visibility / invisibility in consumer UIs is consistent. Further, operators and third party developers can add their own providers afterwards, but phone manufacturer can control where that happens.

[0048] FIG. 4 illustrates a generic parameters implementation in accordance with an exemplary embodiment. In this implementation, a variable called SemanticID is used to describe data, such as FILENAME, URL, MESSAGEID, CONTACTID, IMAGEFILE, IMAGEDATA, and IMAGEHANDLE needed by use cases. The ID has unique value. Closely related SemanticIDs can be grouped together to form group of alternative representations. For example, IMAGEFILE, IMAGEBUFFER, IMAGEDBHANDLE above can be grouped together to

form IMAGE.

[0049] A variable called VariantTypeId is used to represent basic data types, such as integer, date/time, unique ID, string, real, double, and databuffer. TVariant can hold all the data types specified by VariantTypeId. A generic parameter GenericParam owns the following data attributes: SemanticID, Tvariant, SemanticIDGroup, ParameterStatus (which holds status code for data checks), and other utility members for the generic parameter, e.g. originator application UID, version information, and error code. External and internal support for the GenericParam is included to write into a stream and to construct itself from a stream. A stream can represent any operating system stream (memory, file, socket, serial port, IPC channel, etc) which can be used communications needs inside mobile device or between mobile devices.

[0050] A list of generic parameters called GenericParamList is defined which owns one or more GenericParams. External and internal support for the GenericParamList to write into a stream and construct itself from the stream. GenericParamList can be used in every generic function needing consistent way to pass arguments:

Int genericDoSomething(GenericParamList inParams);

Int genericDoSomething(GenericParamList inParams, GenericParamList outParams)

[0051] By way of example, the generic parameters implementation can be used to pass input and output arguments between terminal software product applications using application embedding (in-process communication). As such, the parameter's meaning (semantics) is understood no matter how deep the call stack is. The implementation can also pass input and output arguments between stand-alone

terminal software product applications (process-to-process communication). As such, the semantics and origin of the argument are preserved.

[0052] Moreover, the generic parameters implementation can pass arguments within the Application Interworking Framework 22 described with reference to FIGs. 1-3 above. The AIWFW 22 can utilize the generic parameters to implement data flexible agreements. For example, a service provider command can be executed as follows:

```
void ExecuteServiceCmdL(
        const TInt aCmdId,
        const GenericParamList aInParamList,
 GenericParamList aOutParamList);
```

[0053] As another example, the generic parameters implementation can be used for alternative representations of the semantically same data. For instance, an image container may be file, an identifier of an image database, or image data as memory buffer.

[0054] Referring now to FIG. 4, a CGenericParamList class 52 contains multiple (0..N) CGenericParam items 54. Each CGenericParam item 54 contains a TGenericParamID and TVariant, which are part of a CGenericParam class. The CGenericParam class owns a TVariantTypeID data type class and various data values contained in a TVariant class 56. The CGenericParamList class 52 and the CGenericParam class write to a stream 58 contained in an operating system. An application 60 creates the CGenericParamList class 52 using a Data Utility API 62 which interfaces with a data utility 64 in the creation process.

[0055] In an exemplary embodiment of the creation of CGenericParamList class 52, various classes are defined and identifiers allocated. For example, a class called TGenericParamId is defined to represent SemanticID. The identifiers used in this class are allocated suitable ranges and made unique. A class called TVariantTypeId is defined to represent VariantTypeId. The identifiers used in this class are allocated suitable values and made unique. Variables are also defined for operating system types, such as TInt32, TUid, TTime, TDesC and HBufC. A class called TVariant is defined which owns TVariantTypeId and the actual data in the format above. Further, additional utility attributes are define as member methods or as a CGenericParamList member.

[0056] A class called CGenericParam is defined which has a TGenericParamId data member (iSemanticId), a TVariant data member (iValue), a reserved member for extensions. External and internal methods are provided for the CGenericParam class to write itself into a stream and construct itself from a stream. A class CGenericParamList is defined for the GenericParamList. External and internal support is provided to the list class, too. Basic iterators are provided for listing or accessing the CGenericParamList. A module tester is provided for the classes.

[0057] In alternative embodiments, advanced iteration methods are provided for the class CGenericParamList to search the list by TGenericParamId and TVariantTypeId. The list can then contain alternative presentations of the same parameter. Further, semantic identifier groups can be implemented as range of TGenericParamIds. A semantic identifier can be converted using conversion utilities.

[0058] The exemplary embodiments of the generic parameters implementation enable creation of generic service APIs

which allow handling several kind of arguments without changing function signature. The embodiments also enable semantic checks on arguments checks within a call stack inside and outside process boundaries. A framework type of APIs can apply consistency, security etc. checks based on semantic IDs or even replace one semantic ID with another, more suitable one.

[0059] A check can be made of the semanticID and if needed, one semantic identifier can be converter to another. If RAM consumption is not a problem, caller can accept/create several alternative semantic IDs for same data (e.g. IMAGEFILE, IMAGEDATA, IMAGEHANDLE). Arguments can keep context information based on additional utility data.

[0060] If using in/out arguments, CGenericParamList can also be used as meta-data to describe data agreement required by callee. The caller fills the TVariant part of the data as actual output parameter. The CGenericParamList can be saved into any kind of stream and still keep the semantics in live. The generic parameters implementation extends existing APIs by adding incrementally new semantic identifiers and preserving binary and data capability with earlier implementation.

[0061] FIG. 5 illustrates a device 70 having a central processing unit (CPU) 72, an input 74, an output 76, a memory 78, and a user interface (UI) 79. The UI 79 can be configured with new features according to the exemplary embodiments described with reference to FIGs. 1-4. The CPU 72 processes the instructions contained in the application interworking framework to interface a consumer application and a provider application. The device 70 can be a phone, a personal digital assistant (PDA), a computer, or any other device.

[0062] This detailed description outlines exemplary embodiments of a method, device, system, and a computer program product for dynamically adding features to a software application. In the foregoing description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It is evident, however, to one skilled in the art that the exemplary embodiments may be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to facilitate description of the exemplary embodiments.

[0063] While the exemplary embodiments illustrated in the Figures and described above are presently preferred, it should be understood that these embodiments are offered by way of example only. Other embodiments may include, for example, different techniques for performing the same operations. The invention is not limited to a particular embodiment, but extends to various modifications, combinations, and permutations that nevertheless fall within the scope and spirit of the appended claims.